# A MODULAR COMPILER STRUCTURE : ITS DESIGN AND IMPLEMENTATION FOR C

A Thesis Submitted

in Partial Fulfilment of the Requirements

for the Degree of

**MASTER OF TECHNOLOGY**

by

**K. Venu Madhav**

to the

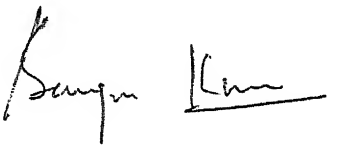
**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**INDIAN INSTITUTE OF TECHNOLOGY, KANPUR**

April, 1992

# CERTIFICATE

It is certified that the work contained in the thesis entitled **A MODULAR COMPILER STRUCTURE : ITS DESIGN AND IMPLEMENTATION FOR C**, by **K Venu Madhav**, has been carried out under my supervision and has not been submitted elsewhere for a degree.



(Dr. Sanjeev Kumar Aggarwal)

Assistant Professor

Dept. of Computer Science and Engg.

I I T, KANPUR

April, 1992.

## Abstract

Every compiler has the front end and code generation phases, which are the minimum essential phases. The optimization phases are optional phases, which enhance the run-time performance of the generated code. The amount of improvement in the run-time performance, caused by an optimization method, and the cost of development of this optimization phase together constitute a performance measure of this optimization method. This measure is called the quantitative performance measure (QPM) of the optimization method. The QPMs aid the compiler engineer in meeting the performance requirements of a compiler. To find the QPM of an optimization method or a sequence of optimization methods, it should be possible to add/remove this optimization method or sequence of optimization methods to/from the compiler, without any modification to other phases. This is only possible with a modular compiler, where the phases are sufficiently alienated and the intermediate representation (IR) is the only link between any two phases. The structure of this modular compiler is designed. The front end of this modular compiler is implemented for a language C.

The work in this thesis

is dedicated

in fond memory of my mother

SATYAVATHI

# ACKNOWLEDGEMENTS

I have a great pleasure in thanking Dr. Sanjeev Kumar Aggarwal for his guidance during the course of this project.

I thank everybody who, and everything which, helped me in making my outlook at the area of computer science broader and broader, during my M.Tech.

I thank all those, particularly the junta in H-top, who made my stay here memorable.

Many thanks to Subba for his help in preparing the figures in this thesis. Thanks to Venku for his help in formatting and printing a part of this thesis.

# TABLE OF CONTENTS

# LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

The task of compiler construction essentially requires two inputs, the source language specification and the target machine specification. The developed compiler has to meet the requirements of functionality, where no compromise is made, and of performance. The various measures of compiler performance are compile-time performance (compiler speed), run-time performance of the generated code (compiler efficiency in generating good code), and good error diagnostics.

The input set along with the performance requirements to be met, influence the development of the compiler. The front end and the code generation phases of a compiler are the minimum essential phases of a compiler (the basis set of the phases). All other phases (optimization phases) are optional phases, which enhance the run-time performance of a compiler.

It can be intuitively said that the improvement in the run-time performance of the generated code, caused by each of the optimization phases is different. It can also be said that different sequences of optimization methods from a set of optimization methods give different amounts of improvement in the run-time performance of the compiler. Hence, different permutations of the optimization methods give different degrees of run-time performance.

The **Quantitative Performance Measure (QPM)** of an optimization method is a set of values defined as follows:

(1) A compiler performance figure (such as percentage of better code, or ratio of compiled code vs. hand code) with the phase which does this kind of optimization as part of the compiler.

(2) The same compiler performance figure without this phase in the compiler.

(3) The cost of development of this phase.

The QPM of a sequence of optimization methods is similarly defined. In the following discussion, the phrase QPMs is used to mean QPMs of different permutations of the optimization methods.

To design and develop a compiler which meets the given run-time performance requirements, the present day compiler engineer has to depend on his experience, as the current literature does not provide him the QPMs. The performance figures published are raw performance figures such as number of source lines compiled per minute, ratios such as 2:1 for compiled code vs. hand code, or percentages such as 10% better code. No light is thrown on how they are realized. It is not known what all contributed to the 10% improvement in code quality, how much compilation effort is required by each code improvement method, and what are the dividends realized from different code improvement methods.

## 1.1 MOTIVATION

The lack of the knowledge regarding the QPMs and the strong belief that the availability of such knowledge helps the compiler engineer realize his run-time performance requirements easily, is the motivation for this work.

The compile-time performance goes down as the number of phases increases. The increase in number of phases may result in increase in number of passes. With more phases and passes, the compile-time performance decreases. The cost of the compiler goes up with increase in number of phases. So, it is advisable to keep the number of phases as low as possible. The QPMs can be used to weed out the optimization methods which give less improvement in the run-time performance but cost considerably high. This results in decrease in cost of the compiler without much decrease in performance.

If each kind of optimization is done by one phase, which can be plugged in and removed out of the compiler without any modification to the remaining phases of the compiler, and if the compiler engineer has such phases for all the optimization methods, with the QPMs at his disposal, the compiler engineer can provide different compilers for different users' requirements (Here requirements mean all the performance requirements. The performance requirements other than run-time performance requirements are met following the principles which are out of the scope of this thesis. The run-time performance requirements are governed by the QPMs). The user needs can now be given as

another input parameter, along with the source language specification and target machine specification.

## 1.2 HOW ARE THE QPMs OBTAINED?

To get the QPMs we need to design and develop such a compiler. The modules of this compiler are the phases of the basis set plus all the phases that provide various kinds of optimization, with each phase providing each kind of optimization. Such a compiler structure is discussed in [KVN]]. This modular structure is discussed in detail in chapter 2.

For this thesis we are developing a front end of a compiler for C (targeted at Sun-3/60 series of workstations (MC68020 processor) and a run-time environment of SunOS (an enhanced version of 4.2 BSD Unix)), around this modular structure. The implementation details are given in chapters 3 and 4.

The conclusions on this work and directions for further work are given in chapter 5.

# CHAPTER 2

# THE MODULAR COMPILER STRUCTURE

It is mentioned earlier, that the availability of the quantitative performance measures (QPMs) would help the compiler engineer realize his performance requirements. A brief outline of how to find these quantitative performance measures is given.

We need a modular compiler to study these quantitative performance measures. The structure of such a compiler should be such that the units with different functions of the compiler are clearly demarkated. This demarkation also makes the compiler development process easier, as various tools, such as scanner generators, parser generators etc., can be used to develop these alienated phases. Such a structure is discussed in [KVN1], and that structure with few modifications is used here. This structure is shown in Figure 2.1, depicting the major segments of the compiler.

The **front end** scans the input source program and then parses it to produce an **intermediate representation** (IR), normally a tree, of the input. Semantic analysis is then performed to get an IR with static type checking done.

The translations from source language entities to target machine entities are done next. The **translate phase** effects semantics-preserving mappings between source language entities and target machine primitives.

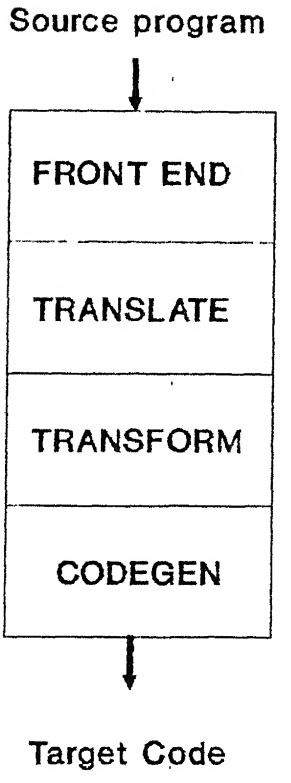



Fig. 2.1 THE MODULAR COMPILER STRUCTURE

Optimizing transformations are performed next by the **transform phase**. These include control-flow analysis, data-flow analysis, target independent optimizations and target dependent optimizations.

The transformed IR is then expressed as a series of machine language statements by **codegen phase**.

## 2.1 THE FRONT END PHASES

Much research has underwent into the front end phases and these have been extensively formalized, making them almost standard [JEH].

The first of these, the **scanner** takes the source program as input and emits the tokens. This phase of the front end is rarely hand coded and the most widely used tool for generating this is **Lex** [MEL]. Another tool which generates a scanner is **flex** [GNU].

The next phase is **parser** which takes the tokens emitted by the scanner and produces an intermediate representation (IR) of the input program, which is normally hierarchic. It checks whether the stream of input tokens confirms to the syntax of the language. If so, it produces the IR, which is syntax error free. If the token stream does not confirm to the syntax of the language, it generates appropriate error messages. This phase can be generated by a tool, **YACC** [SCJ], which is one of the most widely used tool. Another tool which generates a parser is **bison** [GNU].

Static **semantic analyzer** is the next phase of the front end, which does static type checking. This phase resolves ambiguities due to overloading, does type coercion and produces an unambiguous, semantically clean IR.

The front end along with all of its phases is figuratively depicted in Figure 2.2.

## 2.2 THE TRANSLATE PHASES

The translate phase effect semantic-preserving mappings from source language entities to target machine primitives. In this phase, source level data types are converted to machine level

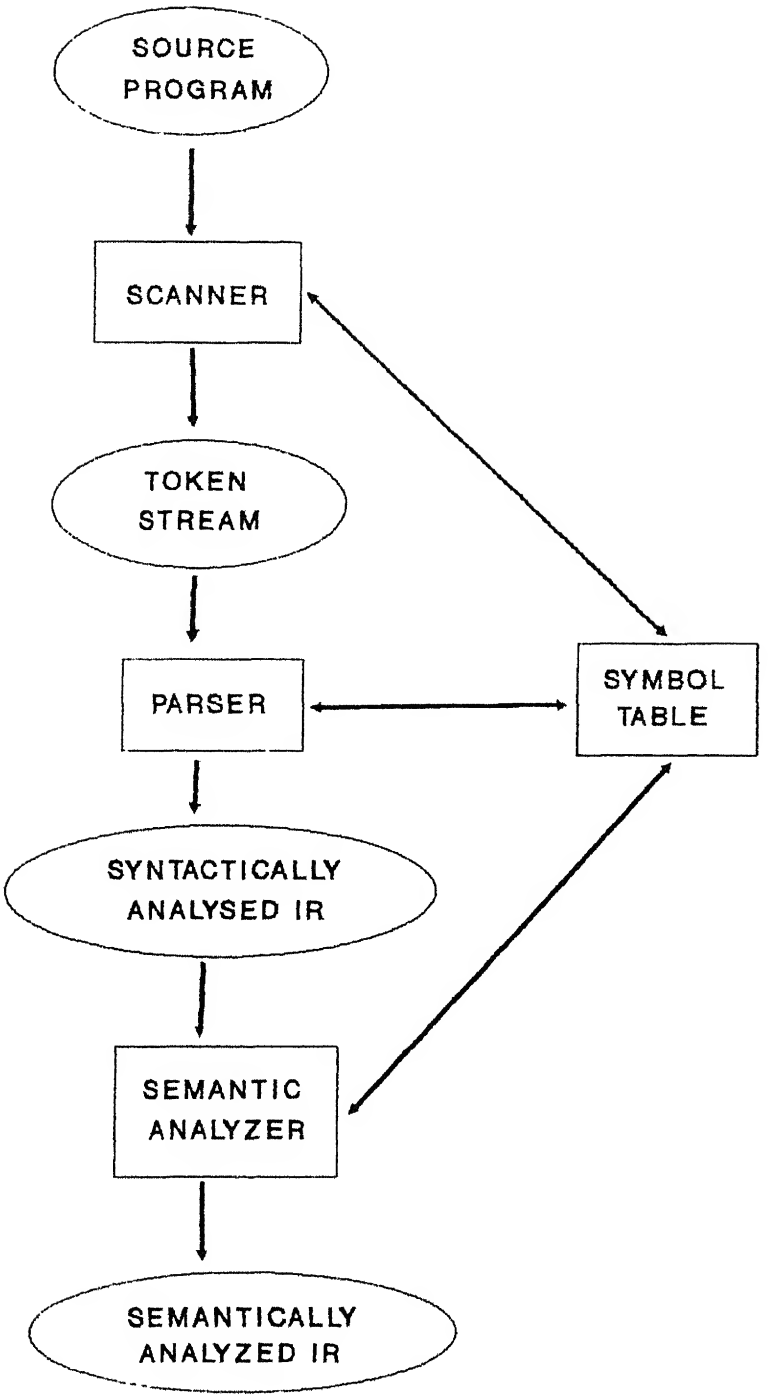



Fig. 2.2 THE FRONT END

data, operators on these data types are converted into machine level functions, implicit aspects of source language are made explicit, and aspects unspecified by the language are performed according to some convention. The resultant IR is such that the code generation can be done by just traversing the IR.

The translate phase can be thought of as the code generation phase of a **virtual target machine (VTM)**, which is at a higher level than the target machine. The generation of the code for this VTM (i.e., semantics-preserving mappings from source language entities to target machine primitives) is trivially done for the primitive value types and operations in source language, which are identical (or easily approximated) to those in VTM. It is done according to a policy in case of an abstraction or a structure. These polices should be context free, so that the mapping of an abstraction or a structure should be the same for all its occurrences in a source program, irrespective of the context of occurrence.

The mappings done by the translate phase can be seen as a sequence of three phases --- **map abstractions**, **map structures**, and **map primitives**. This is depicted in Figure 2.3.

## 2.2.1 MAP ABSTRACTIONS

**Map Abstractions** is a mechanism which maps all abstractions, data and procedural, to simpler structures of the VTM. Data abstraction include the definitions of the data structures and their use. Mapping data abstractions mean, the layout of the data

structures, global and local, in memory, and the mapping of the access mechanisms to access the components of these structures, and the mapping of global and local variables. Procedural abstractions include procedure call-and-return mechanisms, and parameter passing mechanisms. Mapping procedural abstractions mean, mapping the procedure call-and-return mechanisms to control jump instructions of VTM, making a copy of the values or addresses of the actual parameters according to the parameter passing mechanism, and layout of stack frames for recursive procedures.

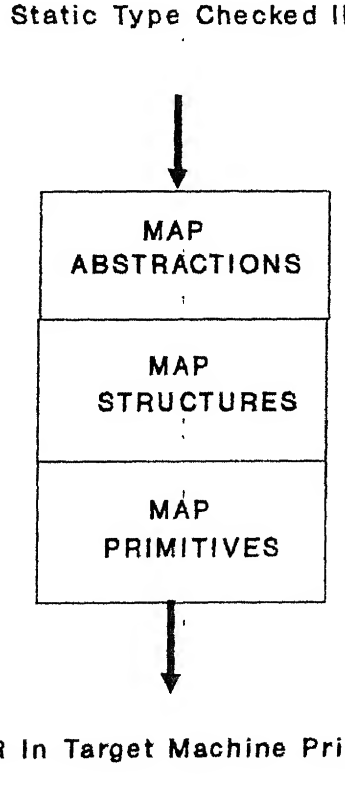



Fig. 2.3 PHASES IN TRANSLATE

### 2.2.2 MAP STRUCTURES

**Map Structures** maps the control structures in source program to simple control flow primitives in the VTM. To do so, it marks the control flow governing expressions and statements of the source program. Then it chooses the appropriate control flow primitive of the VTM to realize these expressions and statements. For example, case statement is realized by indexed jumps. It also generates short life-time temporaries to store the values of variables which are needed throughout the scope of the control structure.

### 2.2.3 MAP PRIMITIVES

**Map Primitives** maps the primitive values and primitive operations in the source program to their equivalents in VTM.

## 2.3 THE TRANSFORMATION PHASES

The transformation phases perform the optimization transformations on the IR which is translated by the translate phase. These optimization transformations result in the generation of code which will be better than the code generated without performing these optimization transformations. The optimization transformations can be grouped into three phases --- **control flow analysis**, data flow analysis, and **code-improving optimization methods**. The optimization methods, such as, common sub-expression elimination, constant folding, tail recursion elimination, code motion, induction variable elimination,

strength reduction are grouped into the code-improving optimization methods phase. Each of these optimization methods is done by one phase. The phases performing all these optimizations are the sub-phases of the code-improving optimization methods phase. Figure 2.4 depicts the model of the transform phase, performing all these transformations in a particular sequence. All these transformations are independent of the target machine architecture.

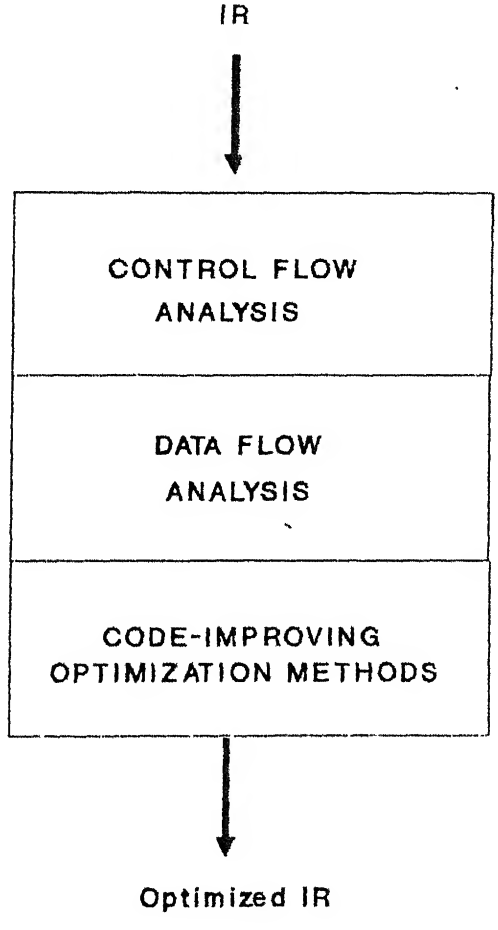



Fig. 2.4 PHASES IN TRANSFORM

## 2.4 THE CODE GENERATION PHASES

The code generator should be retargetable, i.e., for a given source language specification, with few modifications to this phase, this phase should become the code generator of a different compiler targeted at different machine. As we discuss here about a class of compilers rather than a single compiler, this is going to be an important point.

The code generation phase takes the VTM relative IR from the previous phase, traverses it and then generates a sequence of machine language instructions. This is done by **selecting** the machine language instructions from the instruction set of the machine. This sequence of machine instructions should be consistent. That is, when run, this sequence of machine instructions should perform what its source program intends, and nothing else.

It is mentioned above that, to realize better code, optimization transformations are to be performed on the IR before the code is generated. The machine independent optimization transformations are discussed above. The code generation phase may perform the machine dependent optimization transformations, before code selection. These optimizations could be **shape**, which restructures the IR such that a particular traversal of the IR becomes the best traversal, and **allot**, which attempts to do the optimal allotment of the resources. Thus the code generation can be seen as consisting of the three phases --- **shape**, **allot**, and **select**. The Figure 2.5 below depicts a model of code generator

which fuses these three phases sequentially. A detailed discussion on all these three phases is made in [KVN2].

### 2.4.1 SHAPE

Shaping the program consists of restructuring the IR such that a particular traversal of the IR becomes the best traversal. Though it is said above that this phase performs machine dependent optimizations, shaping still undertakes some machine-independent restructuring also, which is based on algebraic properties of operators.

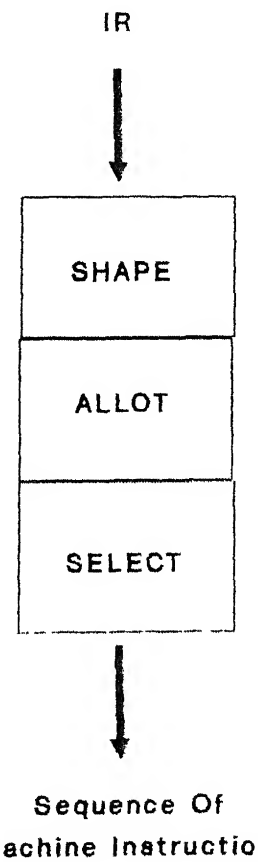



Fig. 2.5 PHASES IN CODEGEN

The machine independent restructuring makes use of the associativity, commutativity, and distributivity properties in the transformation of trees, to effect reduction in the demand for resources, reduction in lifetimes of temporary resources, and better compaction of opcodes, respectively.

The machine dependent restructuring performs code selection using different traversals of the IR, compares them according to some metric, and records the preferred order of traversal.

### 2.4.2 ALLOT

Allot performs global resource allocation so as to guarantee that the remaining resources can be left for the selection phase to allocate them on-the-fly resulting in further improvement of code.

### 2.4.3 SELECT

Retargetable code selection requires the separation of the code selection algorithm from the specification of the target machine instruction to be selected. The code selection is an ambiguous process, for more than one instruction sequence can be locally selected to code the same VTM relative IR. Local optimality in instruction selection is achieved only when the best sequence, with respect to some metric, is selected. This metric could be minimal number of registers needed, or shortest instruction sequence. There are methods available in literature, for performing code selection with respect to some metric. [KVN2] gives their experience using one of these methods.

## 2.5 THE INTERNAL REPRESENTATION

The **internal representation** (IR) of the source programs is normally hierarchic and reflects the structure of the source language compositions used in the input program. The property of the IR is that it has implicit one-in, one-out flow. Explicit control-flow primitives, labels, gotos, breaks, continues and returns violate this property of the IR. The IR should be restructured when these are present in the input program, so as to retain the one-in, one-out property. The effect is to localize the need for iterative methods of data flow analysis.

The phases of the compiler are all threaded by the IR which flows through them. The IR flowing through the successive phases of the compiler undergoes various changes. New operators may be induced by the translate phases. The shape phase may restructure it. All phases may decorate it with attributes.

To make the IR sharp enough for semantics to be directly derived from the structure and content of the IR, a principle is applied to the specification of the IR, while it passes through the sequence of phases. Attributes evaluated to guide any of the compilation activity will be local to a phase. Only the results of the computation will be directly reflected in the structure. This property of the IR implies that there should be only a single thread connecting one phase and its next phase, the IR.

## 2.6 INTEGRATION OF PHASES

We discussed a compiler with phases clearly separated and specified according to their functions. Each phase embodies a single logical activity of the compilation. Once these separate phases are developed, they should be integrated so as to construct passes of a compiler. Such integration may cause the compile-time performance of the compiler to go down. The compiler engineer should take sufficient measures to avoid this.

## 2.7 A FAMILY OF COMPILERS

The modular compiler structure required to measure the QPMs is developed and discussed above. In accordance with our requirements, the phases and sub-phases of this structure are separate enough that they can be easily removed out of and plugged in to the compiler. A compiler engineer can now easily select the phases he need, using the QPMs of these phases, and arrive at a compiler structure which suits his requirements.

The transformation phase in the above structure is optional. The sub-phases shape and allot of the codegen phase are also optional. Figure 2.6 gives a family of compilers in the form of the lattice of the phases of the compiler. The minimum essential phases/sub-phases of a compiler are front end, translate and select. Various combinations of the optional phases can be plugged in to get various compiler structures. The whole of the transformation phase is optional and hence none or all of its sub-phases, or different combinations of them can be used to

achieve the specified degree of optimization.

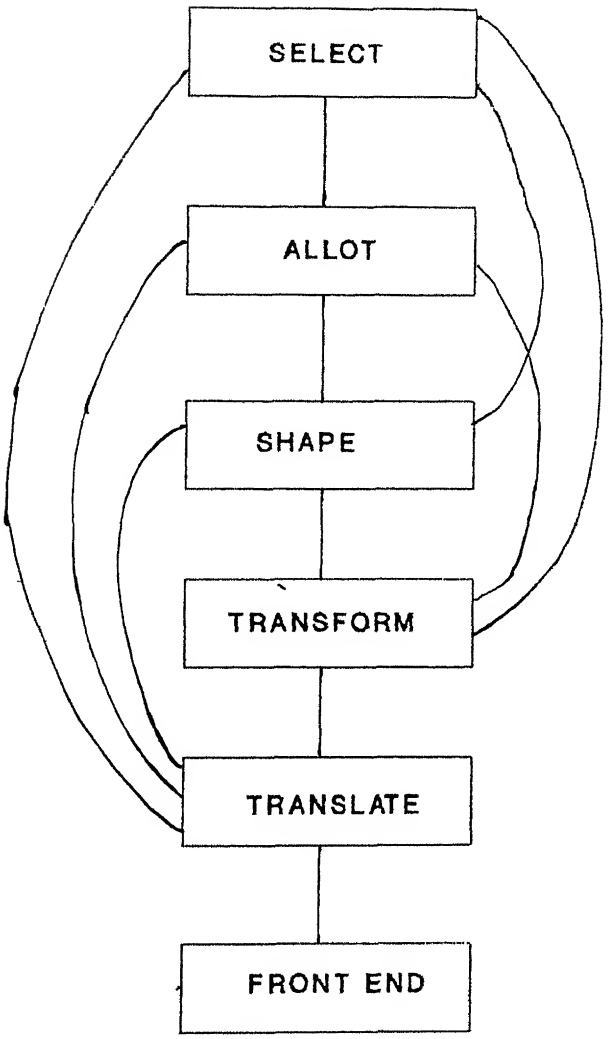



Fig.2.6 A FAMILY OF COMPILERS

### 2.7.1 A FAMILY OF TRANSFORM PHASES

Figure 2.6 suggests that the transform phase can be in or out of a path from front end to select. When this is in a path, it is not that all the sub-phases of the transform phase are included in the path, as these sub-phases are optional. Any of these sub-phases can be in the path in any sequence. So, the sub-phases of the transform phase realize a family of transform phases, whose spectrum ranges from zero sub-phases to all sub-phases in any sequence. In Figure 2.6, the transform phase should be viewed as this spectrum rather than as a single entity.

# CHAPTER 3

# THE LEXER

# AND

# THE SYNTAX ANALYZER

We are developing a compiler around the structure discussed in chapter 2. The source language specification, to this compiler is a language, C, which is defined in appendix A of [BWK]. The target machine specification, is the Sun-3/60 series of workstation (with a processing unit MC68020). The run-time environment of the compiled programs is SunOS.

This chapter discusses the implementation of the lexer and the syntax analyzer. Figure 3.1 below depicts the various phases developed and the tools used in the development of these phases.

## 3.1 LEXER

This phase takes the source program as the input and emits tokens as the output. The syntax analyzer calls the lexer whenever it needs a token and the lexer scans the input source program for a token and returns it. Before returning the token, the lexer may perform some actions, which depend on the the token returned. For example, the action performed before returning the token for a constant is to store the value of this constant in a global variable, to be used by the syntax analyzer.

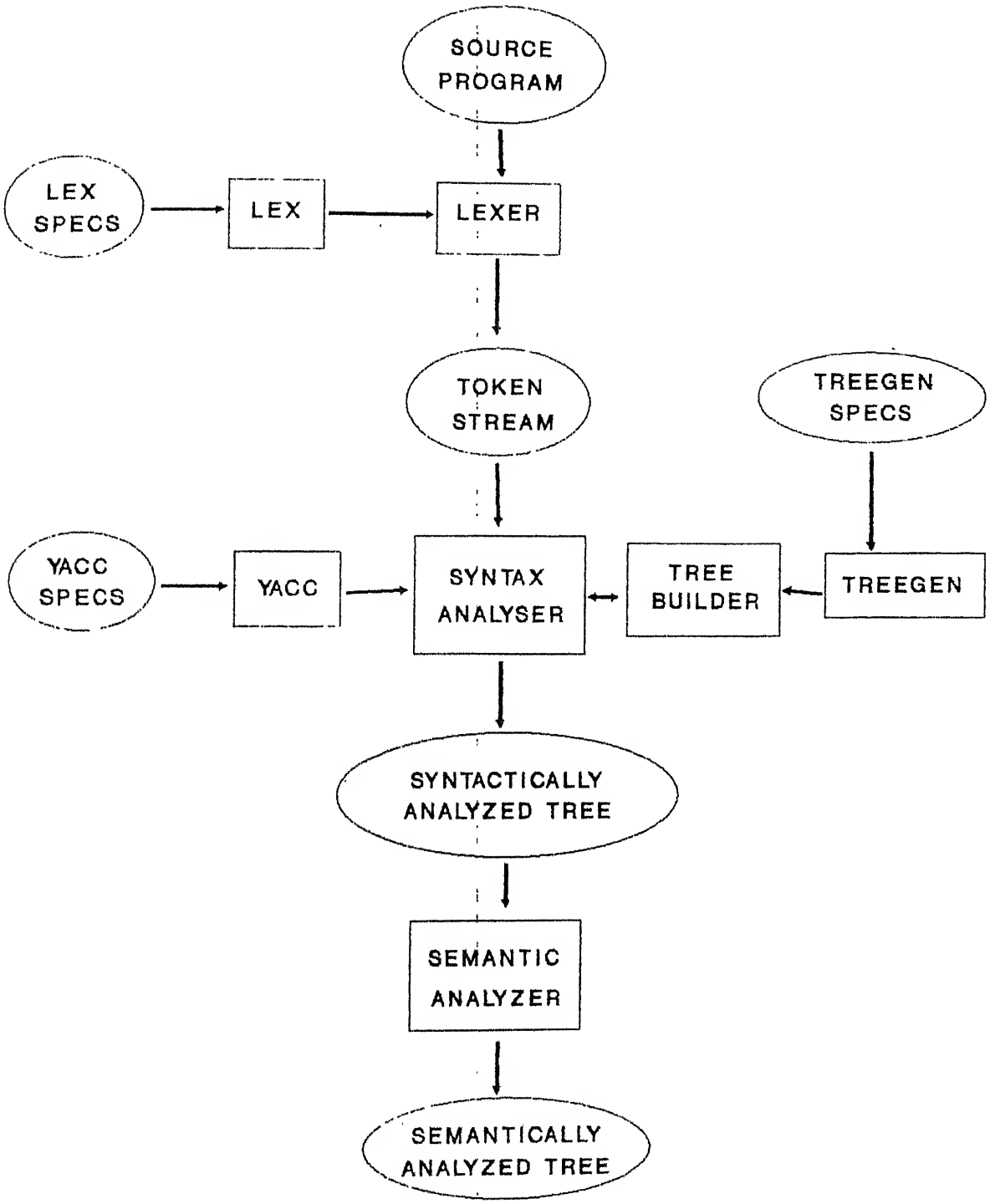



Fig. 3.1 THE FRONT END

The tool, Lex [MEL] is used to generate the lexer phase. The Lex takes a **lex specification** as input and generates the lexer for this specification. This specification is a list of rules. The rules list is a table with left column containing regular expressions and the right column containing actions, program fragments to be executed when the regular expressions are recognized. To generate the lexer for C, the regular expressions for all the possible token strings in C are listed and the actions that are to be performed when the regular expressions are recognized, are specified for each of them. This specification is passed to **Lex** which gave our lexer. This lex specification without the actions is given as appendix A.

## 3.2 SYNTAX ANALYZER

The syntax analyzer takes the token stream given by lexer and checks whether this confirms to the syntax of our language, C. The syntax of the language is specified as a context free grammar (CFG) [AVA]. If the token stream does not reduce according to this CFG, then appropriate error messages are given. If the token stream reduces to the start symbol of the CFG, then the input program is syntactically correct. During this reduction a sequence of actions is executed. These actions construct a tree from the token stream, which is a representation of the input program used by the next phases of the compiler. This tree is constructed bottom-up as the input program is scanned left to right.

The tool, YACC [SCJ] is used for constructing the syntax analyzer. This tool takes an input specification called the **yacc specification** and generates a program and some tables, which together constitute the syntax analyzer for the specification. YACC generates the same program for all kinds of yacc specifications, and the tables generated differ from one yacc specification to other. A yacc specification consists of a set of LALR(1) grammar rules [AVA] and actions are associated with each grammar rule. The actions are arbitrary C statements. The LALR(1) grammar of our language, C, and the actions associated with each rule of this grammar are passed to YACC which gave our syntax analyzer. This grammar is given as appendix B.

## 3.3 TREE BUILDER

The syntax analyzer requires a set of tree building routines to build the intermediate tree representation of the input program. A tool called **Treegen** [TGM] is available which generates these routines.

The Treegen takes a specification of the types of nodes and generates three programs, a **tree builder**, a **tree unparser** and a **tree transformer**, and some tables used by these. As the names of these programs suggest, the tree builder has the routines required to build a tree, the unparser has the routines to unparse a tree, and the transformer has routines to transform a tree. Like YACC, the Treegen also generates different tables for different specification, whereas the routines in tree builder,

tree unparser, and tree transformer does not change. Not all applications require all the three programs generated by Treegen. We can use any of them according to our requirements. Here, only the tree builder is used in this front end.

To facilitate this, the specification to a Treegen is made into various sections, where not all the sections need be specified. Only NODE section is mandatory. Then, if tree builder is required, the FUNCTION and CLASS sections can be specified, if tree unparser is required, unparse specifications can be included in these sections, and if tree transformer is required, VARIABLE and RULE sections can be included.

### 3.3.1 THE NODE STRUCTURE

Three kinds of nodes can be used in the construction of trees using treegen, a **leaf node**, a **list node**, and an **other node**. The leaf node, as the name suggest, is a leaf node in the tree and the other two are internal nodes. The structure or **type** of a node is characterized by its name, its kind, number of sons (for list and other nodes), and the types of the sons (for list and other nodes). The **treegen specification** contain the various types of nodes in the tree.

A node in the tree, built by the tree builder has the structure given in Figure 3.2.

The name of the language construct, for which the node is created, is stored in **nodetype** field of the structure. The line number in the input source program, in which this language

construct is present, is stored in **lineno** field. **Info** is any information regarding this node. For example, the number of sons of a list node can be stored in this field. The information stored in the **type** field of this structure depends on the kind of the node. For a leaf node, the information stored is the string identifying the leaf (leafid). For a list node, a pointer to the linked list of the sons is stored (listson). For a other node, an array of pointers to the sons is stored (son).

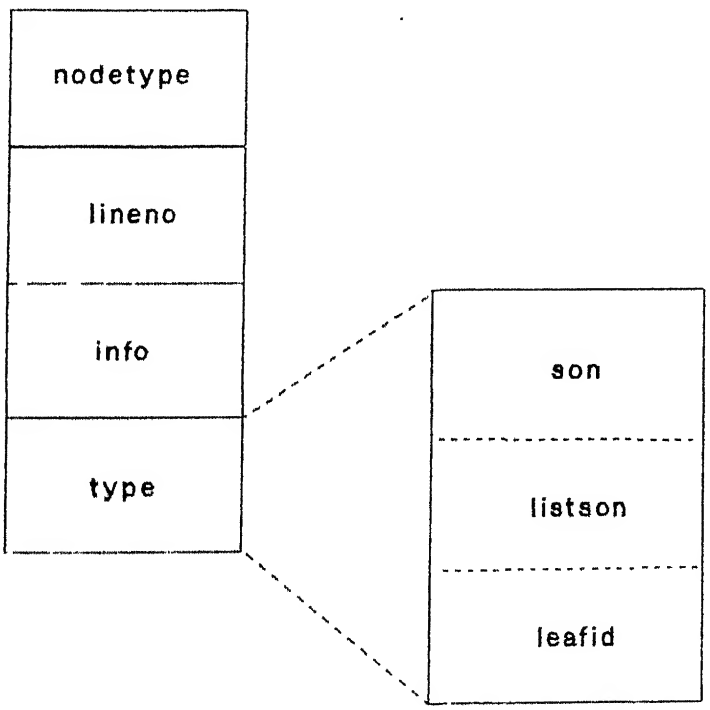



Fig. 3.2 NODE STRUCTURE

### 3.3.2. THE DESIGN OF IR

For each construct of the language C, the type of the node to represent this construct in the IR is decided. For example, the type of node of **for statement** is as in Figure 3.3.

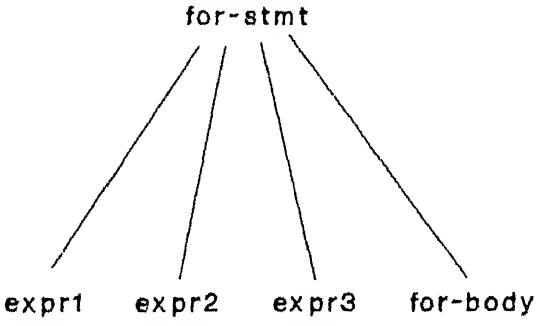



Fig. 3.3 NODE TYPE OF FOR STATEMENT

The treegen specification is written for all the language constructs, which is a list of the types of nodes for these constructs. For example, the treegen specification of **for statement** specifies that node representing **for statement** in the IR has four sons, the type of first three being the class expr and the type of the last son being the class statement.

```
for_stmt  :  < first_expr  : /expr/,
               second_expr : /expr/,
               third_expr  : /expr/,
               for_body    : /statement/ >
```

This **treegen specification** is passed to **Treegen** which generates the **tree builder**. The routines **makeleaf()**, and **makenode()** in the generated tree builder are used by the syntax analyzer to build the tree, as the input program is parsed by it. The treegen specification is given as appendix C.

## 3.4 SYMBOL TABLE

The information about all the names in the input C program is stored by the front end phases, so that this can be used later by the other phases. Such information is stored in symbol table. The information stored in symbol table about a name includes the identifier string representing the name, the block number in which this name is defined, the kind (a variable , a function , or a typedef ) of the name, and the kind-specific information.

### 3.4.1 THE SYMBOL TABLE NODE STRUCTURE

The symbol table node to hold the above information is given in Figure 3.4. The string representing a name is stored in the **name** field. The number of the block in which this name is defined is stored in **blk_num** field. The kind (a variable, a function, or a typedef) of the name is stored in **entry_type** field. The kind-specific information is stored in **attribs** field which is a union.

The kind-specific information for a name of kind **variable** is stored in a structure given in Figure 3.4(b). The storage class of the variable is stored in **store_class** field. The type of the variable is stored in **types** field. The declarator information

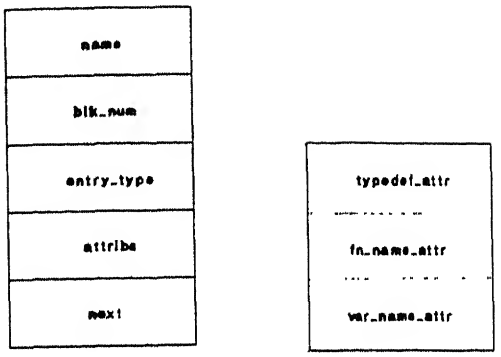



Fig. 3.4(a)

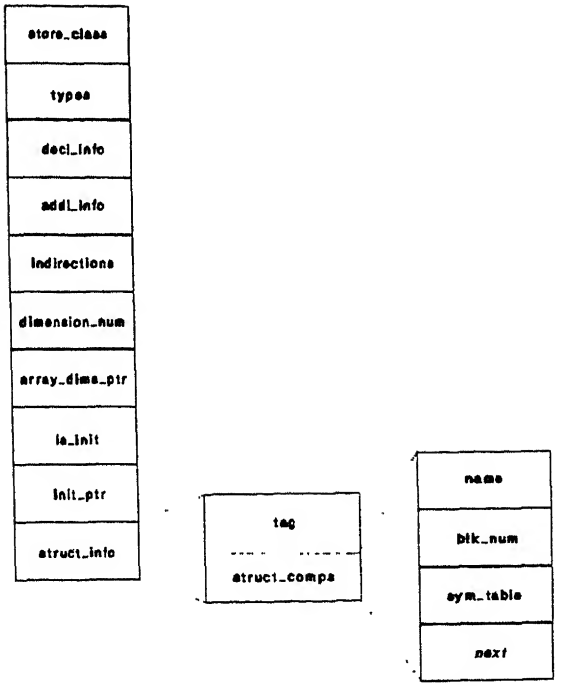



Fig. 3.4(b)

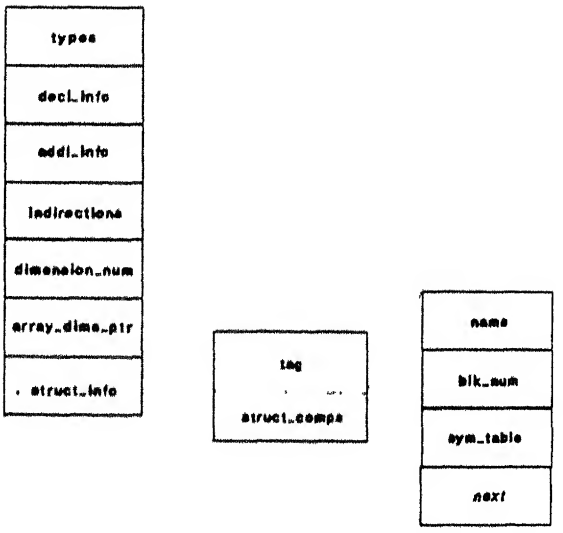



Fig. 3.4(c)

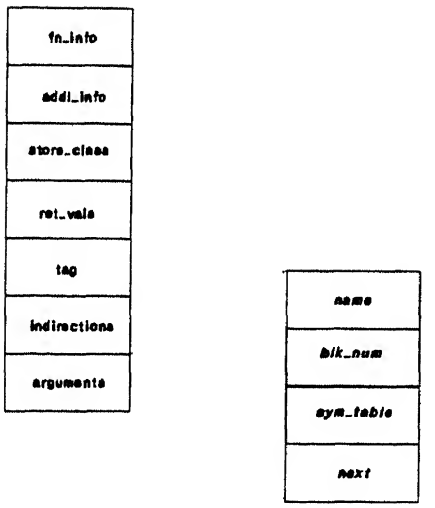



Fig. 3.4(d)

Fig. 3.4 SYMBOL TABLE NODE STRUCTURE

(whether the declarator specifies an array, a pointer, a function, a pointer to function, an array of pointers etc.) is stored in **decl_info field**. Any additional information about the declarator is stored in **addl_info** field. The number of pointer indirections (in cases where declarator specifies a pointer) is stored in **indirections** field. The number of dimensions (in cases where the declarator specifies an array) is stored in **dimension_num** field. The array dimensions can be had via a pointer **array_dims_ptr** which points to a node in IR. The information whether a variable is initialized or not is stored in **is_init** field. The initial values can be had via a pointer **init_ptr** which points to a node in IR. If the type is struct/union, the information about the tag and the components is stored in a structure **struct_info**.

The kind-specific information for a name of kind **typedef** is stored in a structure given in Figure 3.4(c). It is similar to the structure used for storing variable's information expect that it does not has store_class, is_init, and init_ptr fields.

The kind-specific information for a name of kind **function** is stored in a structure given in Figure 3.4(d). The function name information (whether the function name specifies a pointer to a function, or a function returning a pointer) is stored in **fn_info** field. Any additional information about the function name is stored in **addl_info** field. The storage class of the function is stored in **store_class** field. The return value type is stored in **ret_vals**. The struct/union tag is stored in the **tag** field (in

cases where return value is a pointer to a struct/union). The number of pointer indirections (in cases where the function name specifies a pointer) is stored in **indirections** field. The information about the arguments is stored in **arguments** field.

The symbol table is implemented as a linked list. A function **look_up()** is provided to look-up the symbol table for a name and its information. Another function **update()** is provided to add a new entry to the symbol table or to add more information to an existing entry. This implementation makes the use of symbol table easy, as all operations on the symbol table can be performed using these functions.

# CHAPTER 4

# THE SEMANTIC ANALYZER

The implementation of the semantic analyzer is discussed in this chapter.

The semantic analyzer performs static type checking on the intermediate representation (IR) generated by the syntax analyzer. It gives appropriate error messages, if any type mismatches are detected. Otherwise the semantically analyzed IR is passed to the next phase of the compiler for further processing. While the semantic analyzer performs **static type checking**, the dynamic type checking is done during the execution of the generated code. In the following discussion the phrase type checking is synonymously used to mean static type checking and the phrase type checker is synonymously used to mean semantic analyzer.

The appendix A of [BWK] defines the C language. The type checkings are to be performed on the IR so that the input source program confirms to the definition of the language. The various type checkings performed by the front end is given below.

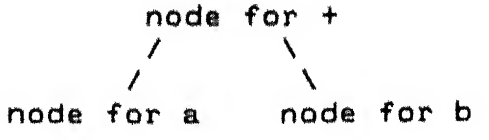
## 4.1 TYPE CHECKING OF EXPRESSIONS

The type checking in expressions is performed using the syntax-directed translation scheme. The attributes of the identifiers and the operators are stored (annotation) in the node

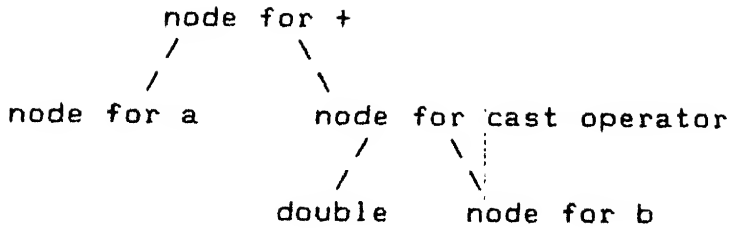
representing them in the IR. The annotation of the type information is done in the semantic analyzer phase. The types of the identifiers in an expression are obtained from the symbol table. The nodes in the IR representing them are annotated with these types. The type checker checks whether the types of these operands confirm to the types accepted by the operator, and generates an error message if not. For example, the operands of the operator **%** must not be **float**. If the operands confirm to the types accepted by the operator, then the type of the result is the type of expression and the node for this expression in the IR is annotated with this type.

In C many operators cause type conversions of operands (C is a weakly-typed language) and yield results depending on these conversions. These conversions are done using cast and this casting is explicitly reflected in the IR. The conversions are given in appendix A of [BWK].

Suppose, if the type of a is **double**, and type of b is **int**, the expression a + b is represented in the IR as follows:




The type conversion rules say that when either operand is **double** the other is converted to **double**, and the type of the result is **double**. The type checker explicitly casts the type of b to **double** according to this rule and modifies the IR to reflect this. The modified IR looks as follows:




The result of the expression is **double** which is stored in the node for +, which represents the expression in the IR.

## 4.2 TYPE CHECKING OF DECLARATIONS

The type checking of the declarations is performed, basically, by storing the information required by the semantic analyzer, in the IR, during the syntax analysis phase, and making use of this information to check that the declarations confirms to the definition of the language, during the semantic analysis phase.

The declarations are checked for consistency. A declaration is consistent if it has at most one valid type specifier, at most one storage class specifier, if the declarators in the declarations are consistent, and if there are definitions of the declarators somewhere, if the storage class specifier is **extern**. A type error is given if they are inconsistent.

All the valid type specifiers are stored in a table. The IR is traversed and when a type specifier is found during the traversal, it is compared with the type specifiers in this table.

The syntax analyzer makes a note of the number of storage class specifiers in a declaration. When this exceeds one, for a declaration, an error message is given.

The IR is traversed to find the declarations with storage class specifier **extern**, and when found, the symbol table is searched for the names in this declarations. Error message is given, if any of these names is not found.

The declarators in the declarations are also checked for consistency. A declarator is inconsistent if it is an array of functions, or a function returning array, structure, union or a function. The legality of function returns is done in action part of the syntax analyzer. The legality of the items of an array is also checked in the action part of the syntax analyzer.

All structure and union declarations are checked for consistency. They are inconsistent if a member declaration is an instance of the parent structure/union, if a member is a function, if a member is an array of fields, or if the names of a member and a struct/union tag are same. The IR is traversed to get the information of the member declarations of a structure/union to check these.

## 4.3 TYPE CHECKING OF EXTERNAL DEFINITIONS

The type checking of the external definitions is done in similar lines in which the type checking of the declarations is done.

All function definitions are checked for consistency. An external function definition is inconsistent if the storage class specifier is other than **extern** or **static**, parameters other than those in the parameter list are declared, or the storage class

specifier of a parameter declaration is other than **register**.

All external data definitions are checked for consistency. An external data definition is inconsistent if its storage class specifier is other than **extern** or **static**.

The block number of a declaration is stored along with the storage class specifier during syntax analysis. The tree is traversed, and block number information of all storage class specifiers is collected. If a storage class specifier with block number zero is other than **extern**, **static**, or **typedef**, then an error message is given.

The list of arguments and the argument declarations are stored in the IR during the syntax analysis. The semantic analyzer traverses the IR and checks whether the declarations match the arguments in the argument list or not.

## 4.4 TYPE CHECKING DONE IN OTHER PHASES

The type checking is not confined to the semantic analyzer alone. Some of the type checking is done during syntax analysis also. Each identifier being declared is checked whether it is declared earlier and if it is, then an error message is given. This is done by syntax analyzer with the help of the symbol table **look_up** routine. During each use of every name, the syntax analyzer checks whether the given name is declared or not. The condition that at most one storage class specifier is allowed in a declaration is checked by the syntax analyzer. Legality of the

function return value is checked by syntax analyzer. Declaration of array of functions are diagnosed by the syntax analyzer.

# CHAPTER 5

# CONCLUSIONS

# AND

# DIRECTIONS FOR FURTHER WORK

The quantitative performance measure (QPM) of an optimization method is defined and a discussion is made on how useful these will be for a compiler engineer. To study the QPMs, the need for a modular compiler is discussed. The front end of such a compiler is developed and its design and implementation are discussed.

The whole C as it is specified in appendix A of [BWK] is implemented. The size of the developed front end is about 5500 lines, the break up being, lex specifications -- about 200 lines, yacc specifications -- about 3000 lines, treegen specifications -- about 200 lines, semantic analyzer -- about 1000 lines, C code for implementing symbol table -- about 500 lines, and data structures' definitions and other C code -- 500 lines.

The testing of this front end is done on inputs, like, the programs generated by Lex and YACC for the specifications used to develop this front end.

## 5.1 DIRECTIONS FOR FURTHER WORK

Appendix D gives a brief notes on the source code files of this front end. This is given to aid those who carry further work.

The immediate work that can be done is to develop the back end of the compiler and get the working compiler. Once the compiler is available, the experiments to study QPMs of various optimization methods can be performed.

The experimental study would be on these lines:

(1) Develop a phase of the compiler which performs one kind of optimization.

(2) Note the cost of development of this phase.

(3) Plug in this phase into the compiler. Obtain various performance measures of a compiler.

(4) Remove this phase from the compiler. Obtain the same performance measures.

(5) Tabulate the set of values found in 2, 3, and 4, as the QPM of this kind of optimization.

Repeat this experiment for all optimization methods.

To study the QPM of sequence of optimization techniques, repeat the above experiment replacing the phase with a sequence of phases.

# REFERENCES


[KVN1] K. V. Nori, S. Kumar, and M. P. Kumar, **Retrospection on PQCC Compiler Structure**, FST and TCS7 conference, LNCS vol. 287, Springer Verlag, 500-527, 1987.

[JEH] John E. Hopcroft, and Jeffery D. Ullman, **Introduction to Automata Theory, Languages, and Computation**, Narosa Publishing House, 1987.

[MEL] M. E. Lesk, **Lex -- A Lexical Analyzer Generator**, Computing Science Technical Report #39, Bell Laboratories, Murray Hill, New Jersey, July 1975.

[GNU] **GNU Free Software**, Free Software Foundation, Inc., 675 Mass Ave., Cambridge, MA 02139, USA.

[SCJ] S. C. Johnson, **Yacc: Yet Another Compiler Compiler**, Computing Science Technical Report #32, Bell Laboratories, Murray Hill, New Jersey, Oct. 1975.

[KVN2] K. V. Nori, S. Kumar, and R. V. Deodhar, **Experience with a Retargetable Code Generator**, Proceedings of the workshop on Compiler Compiler and High Speed Compilation, Berlin, 412-426, 1988.

[BWK] Brain W. Kernighan and Dennis M. Ritchie, **The C programming Language**, Prentice Hall of India Private Limited, New Delhi, 179-219, 1986.

[AVA] Alfred V. Aho, Ravi Sethi, and Jeffery D. Ullman, **Compilers: Principles, Techniques, and Tools**, Addison-Wesley Publishing Company, 1986

[TGM] **Treegen Manual, Version 6.0**, Tata Research Development and Design Centre, Pune.

# APPENDIX A

# LEX SPECIFICATION

In this appendix the regular expressions of the C language used as part of the lex specification, which generated the lexer phase of the front end developed in this thesis, is given.

```
ws              [ \t\n]*
bt              [ \t]*
letter          [a-zA-Z]
digit           [0-9]
id              ({letter}|_)({letter}|{digit}|_)*
dec             (([1-9][0-9]*)|[0])
oct             ([0][0-7]+)
hex             ([0][xX][0-9a-fA-F]+)
intcons         ({dec}|{oct}|{hex})([lL]?)
floatcons       {dec}?(\.[0-9]+)?([Ee][+\-]?({dec}|{oct}|{hex}))?
symbops         [\(\)\{\}\[\]\+\-\*\/\%\.\,\|\&\~\^\!\:\;\?\=\<\>]
normchars       [^\\\']
escchars        \\[ntbrf\\\']
octchars        \\[0-7]{3}
chars           ({normchars}|{escchars}|{octchars})
strliteral      (\"(([^\\\"])|([\\][^\"])|([\\][\"]))*\")
comment         \/\*([^\*]|[\*][^\/]|{ws})*\*\/
preproc         [\#]([^\n])*[\n]


%%
{bt}
"\n"
{comment}
"#"
{id}
"*="
"+="
"-="
"/="
"%="
">>="
"<<="
"&="
"|="
"^="
"||"
```

```
"&&"
"==="
"!="
">="
"<="
">>"
"<<"
"++"
"--"
"->"
{symbops}
{intcons}
{floatcons}
\'({chars})\'
{strliteral}
%%
```

# APPENDIX B

## LALR(1) GRAMMAR FOR C

This appendix lists the LALR(1) grammar of C, used in yacc specification to generate the syntax analyzer developed in this thesis. This grammar is heavily influenced by the routines in the tree builder, used in building the tree (IR).

```
program         :       /*  empty  */
                |       extdefs
                ;

extdefs         :       set_markstack  extdef
                |       extdefs  extdef
                ;

extdef          :       datadef
                |       fndef
                ;

datadef         :       decl
                |       set_markstack_rec  initdecls
                        reset_tempsl
                |       error  ';'  reset_tempsl
                |       error  '}'  reset_tempsl
                |       ';'
                ;

decl            :       set_markstack_rec  typed_declspecs
                        set_markstack  initdecls  ';'  reset_tempsl
                |       set_markstack_rec  typed_declspecs  ';'
                        reset_tempsl
                ;

reset_tempsl    :       /* empty */
                ;

fndef           :       set_markstack_rec  typed_declspecs
                        set_markstack  id_declarator  xdecls
                        compstmt_or_error  reset_temps2
                |       set_markstack_rec  id_declarator  xdecls
                        compstmt_or_error  reset_temps2
                ;
```

```
reset_temps2    :       /* empty */
                ;

set_markstack   :       /* empty */
                ;

set_markstack_rec:
                /* empty */
                ;


/*  expressions  */

expr            :       nonnull_exprlist
                ;

xexpr           :       /*  empty  */
                |       expr
                ;

nonnull_exprlist:
                        expr_no_commas
                |       nonnull_exprlist  ','  expr_no_commas
                ;

expr_no_commas:
                        cast_expr
                |       expr_no_commas  '+'  expr_no_commas
                |       expr_no_commas  '-'  expr_no_commas
                |       expr_no_commas  '*'  expr_no_commas
                |       expr_no_commas  '/'  expr_no_commas
                |       expr_no_commas  '%'  expr_no_commas
                |       expr_no_commas  '>'  expr_no_commas
                |       expr_no_commas  '<'  expr_no_commas
                |       expr_no_commas  EQL  expr_no_commas
                |       expr_no_commas  NOTEQL  expr_no_commas
                |       expr_no_commas  GRTEQL  expr_no_commas
                |       expr_no_commas  LESSEQL  expr_no_commas
                |       expr_no_commas  LOGICOR  expr_no_commas
                |       expr_no_commas  LOGICAND  expr_no_commas
                |       expr_no_commas  '&'  expr_no_commas
                |       expr_no_commas  '^'  expr_no_commas
                |       expr_no_commas  '|'  expr_no_commas
                |       expr_no_commas  RTSHT  expr_no_commas
                |       expr_no_commas  LTSHT  expr_no_commas
                |       expr_no_commas  asgnop  expr_no_commas
                |       expr_no_commas  '?'  expr_no_commas
                        ':'  expr_no_commas
                ;

cast_expr:
                        unary_expr
                |       '('  typename  ')'  cast_expr
                ;
```

```
asgnop          :       '='
                |       APLUSEQ
                |       AMINUSEQ
                |       AMULEQ
                |       ASLHEQ
                |       AMODEQ
                |       ARTSHTEQ
                |       ALTSHTEQ
                |       AANDEQ
                |       AOREQ
                |       AXOREQ
                ;

unary_expr      :       primary
                |       unop  cast_expr
                |       SIZEOF  unary_expr
                |       SIZEOF  '('  typename  ')'
                ;

primary         :       IDENTIFIER
                |       INTCONST
                |       CHARCONST
                |       FLOATCONST
                |       STRING
                |       '('  expr  ')'
                |       '('  error  ')'
                |       primary  '('  set_markstack  xexpr  ')'
                |       primary  '['  expr_no_commas  ']'
                |       primary  '.'  IDENTIFIER
                |       primary  POINTSAT  IDENTIFIER
                |       primary  INCR
                |       primary  DECR
                ;

unop            :       '*'
                |       '-'
                |       '&'
                |       '!'
                |       '~'
                |       DECR
                |       INCR
                ;


/*   declarations  */

xdecls          :       /*  empty  */
                |       decls
                ;

decls           :       set_markstack  decl
                |       decls  decl
                ;
```

```
typed_declspecs:
                typespec  reserved_declspecs
        |       declmods  reserved_declspecs
        ;

reserved_declspecs:
                /*  empty  */
        |       reserved_declspecs  typespec
        |       reserved_declspecs  SCSPEC
        ;

declmods:
                SCSPEC
        ;

typespec        :       TYPESPEC
        |       structsp
        |       TYPENAME2
        ;

initdecls       :       initdcl  reset_temps3
        |       initdecls  ','  initdcl  reset_temps3
        ;

initdcl         :       declarator  '='  set_markstack  init
        |       declarator
        ;

init            :       expr_no_commas
        |       '{'  initlist  '}'
        |       error
        ;

initlist        :       init
        |       init  ','
        |       initlist  init
        |       initlist  init  ','
        ;

declarator      :       id_declarator
        |       typename_declarator
        ;

id_declarator   :       '*'  id_declarator
        |       '('  id_declarator  ')'
        |       id_declarator  '('  set_markstack  parmlist
        |       id_declarator  '['  expr_no_commas  ']'
        |       id_declarator  '['  ']'
        |       IDENTIFIER
        ;
```

```
typename_declarator:
                '('  typename_declarator  ')'
        |       '*'  typename_declarator
        |       typename_declarator  '('  ')'
        |       typename_declarator  '['  expr_no_commas  ']'
        |       typename_declarator  '['  ']'
        |       TYPENAME1
        ;

reset_temps3  :       /* empty */
        ;

structsp      :       STRUCT  IDENTIFIER  '{'  blk_begin
                      component_decl_list  '}'  blk_end
        |       STRUCT  '{'  blk_begin  component_decl_list
                      blk_end
        |       STRUCT  IDENTIFIER
        |       UNION  IDENTIFIER  '{'  blk_begin
                      component_decl_list  '}'  blk_end
        |       UNION  '{'  blk_begin  component_decl_list  '}
                      blk_end
        |       UNION  IDENTIFIER
        ;

component_decl_list:
                set_markstack  component_decl  ';'  reset_temp
        |       component_decl_list  component_decl  ';'
                reset_temps5
        ;

reset_temps5  :       /* empty */
        ;

component_decl:
                set_markstack  typed_declspecs  components
        |       error
        ;

components    :       set_markstack  component_declarator
                      reset_temps3
        |       components  ','  component_declarator
                      reset_temps3
        ;

component_declarator:
                declarator
        |       declarator  ':'  expr_no_commas
        |       ':'  expr_no_commas
        ;

typename      :       set_markstack  typed_declspecs  absdcl
                      reset_temps4
        ;
```

```
absdcl          :       /*  empty */
                |       '('  absdcl  ')'
                |       '*'  absdcl
                |       absdcl  '('  ')'
                |       absdcl  '['  expr_no_commas  ']'
                |       absdcl  '['  ']'
                ;

reset_temps4    :       /* empty */
                ;


/*  statements   */

stmts           :       set_markstack  stmt
                |       errstmt
                |       stmts  stmt
                |       stmts  errstmt
                ;

errstmt         :       error  ';'
                ;

compstmt_or_error:
                        compstmt
                |       error  compstmt
                |       errstmt  compstmt
                ;

compstmt        :       '{'  '}'
                |       '{'  blk_begin  stmts  '}'  blk_end
                |       '{'  blk_begin  decls  stmts  '}'  blk_end
                |       '{'  blk_begin  decls  '}'  blk_end
                |       '{'  error  '}'
                ;

stmt            :       compstmt
                |       expr  ';'
                |       IF  '('  expr  ')'  stmt  ELSE  stmt
                |       IF  '('  expr  ')'  stmt
                |       WHILE  '('  expr  ')'  stmt
                |       DO  stmt  WHILE  '('  expr  ')'  ';'
                |       FOR  '('  xexpr  ';'  xexpr  ';'  xexpr  ')'
                        stmt
                |       SWITCH  '('  expr  ')'  stmt
                |       CASE  expr_no_commas  ':'  stmt
                |       DEFAULT  ':'  stmt
                |       BREAK  ';'
                |       CONTINUE  ';'
                |       RETURN  ';'
                |       RETURN  expr  ';'
                |       GOTO  IDENTIFIER  ';'
                |       IDENTIFIER  ':'  stmt
                |       ';'
                ;
```

```
blk_begin       :       /* empty */
                ;

blk_end         :       /* empty */
                ;


/*    parameter lists   */

parmlist:
                        empty_parmlist   ')'
                |       identifiers   ')'
                |       error   ')'
                ;

empty_parmlist:
                        /*   empty   */
                ;

identifiers     :       IDENTIFIER
                |       identifiers   ','   IDENTIFIER
                ;
```

# APPENDIX C

# TREEGEN SPECIFICATION

This appendix lists the treegen speifications used to generate the tree builder, which provided the routines to build tree (IR), to the syntax analyzer.

```
CLASS

/expr/          :       < comma_expr, binop_expr, asgnop_expr,
                          fn_call_expr, array_ref_expr, pointsat_expr,
                          dot_ref_expr, coercion_expr, sizeof_expr,
                          ident, constant, string, cond_expr,
                          unop_expr >

/statement/     :       < for_stmt, while_stmt, do_stmt, if_stmt,
                          ifelse_stmt, return_stmt, break_stmt,
                          cont_stmt, switch_stmt, goto_stmt,
                          label_stmt, comp_stmt, /expr/, NULL >

NODE

program         :       < [defs_fns     : extdef] >

extdef          :       < declarations  : {decls, NULL},
                          functions     : {fns, NULL} >

decls           :       < [decl         : decl_i] >

fns             :       < [func         : fn_i] >

decl_i          :       < store_class   : {scspec, NULL},
                          types         : type_spec,
                          struct_tag    : {st_tag, NULL},
                          variables     : {vars, NULL},
                          st_defs       : {decls, NULL},
                          st_field      : {/expr/, NULL} >
```

```
fn_i          :   < func_name      : ident,
                    is_pointer     : {true, NULL},
                    indirections   : {indir_level, NULL},
                    ret_values     : type_spec,
                    store_class    : {scspec, NULL},
                    isptr_ret_val  : {true, NULL},
                    retval_tag     : {st_tag, NULL},
                    arguments      : {args, NULL},
                    arg_decls      : {decls, NULL},
                    fn_body        : {comp_stmt, NULL} >

type_spec     :   < [type          : type_spec_i] >

scspec        :   < >

vars          :   < [variab        : var_i] >

st_tag        :   < >

ident         :   < >

true          :   < >

args          :   < [argument      : arg_i] >

comp_stmt     :   < declarations   : {decls, NULL},
                    stmts          : {stmt_body, NULL} >

type_spec_i   :   < >

var_i         :   < variable_name  : ident,
                    is_pointer     : {true, NULL},
                    indirections   : {indir_level, NULL},
                    is_function    : {true, NULL},
                    isptr_to_fn    : {true, NULL},
                    isfn_ret_ptr   : {true, NULL},
                    is_array       : {true, NULL},
                    isary_of_ptrs  : {true, NULL},
                    dimensions     : {dims, NULL},
                    dimension_num  : {dim_num, NULL},
                    init_values    : {init_val, NULL} >

arg_i         :   < argument       : {/expr/, ident} >

stmt_body     :   < [stmt          : /statement/] >

indir_level   :   < >

dims          :   < [array_dim     : dim_i] >

dim_num       :   < >

init_val      :   < [initial_val   : init_val_i] >
```

```
comma_expr      :       < first_expr    : /expr/,
                          second_expr   : /expr/ >

binop_expr      :       < first_expr    : /expr/,
                          second_expr   : /expr/ >

asgnop_expr     :       < lvalue        : /expr/,
                          rvalue        : /expr/ >

unop_expr       :       < expression    : /expr/ >

fn_call_expr    :       < func_name     : ident,
                          arguments     : {args, NULL} >

array_ref_expr:         < array_name    : /expr/,
                          index         : /expr/ >

pointsat_expr   :       < struct_name   : /expr/,
                          member        : ident >

dot_ref_expr    :       < struct_name   : /expr/,
                          member        : ident >

coercion_expr   :       < typename      : type_name,
                          expression    : /expr/ >

sizeof_expr     :       < typename      : {type_name, /expr/} >

constant        :       < >

string          :       < >

cond_expr       :       < first_expr    : /expr/,
                          second_expr   : /expr/,
                          third_expr    : /expr/ >

for_stmt        :       < first_expr    : /expr/,
                          second_expr   : /expr/,
                          third_expr    : /expr/,
                          for_body      : /statement/ >

while_stmt      :       < expression    : /expr/,
                          while_body    : /statement/ >

do_stmt         :       < do_body       : /statement/,
                          expression    : /expr/ >

if_stmt         :       < first_expr    : /expr/,
                          if_body       : /statement/ >

ifelse_stmt     :       < first_expr    : /expr/,
                          if_body       : /statement/,
                          else_body     : /statement/ >
```

```
return_stmt   :   < ret_value      : {/expr/, NULL} >

break_stmt    :   < >

cont_stmt     :   < >

switch_stmt   :   < switch_expr    : /expr/,
                    declarations   : {decls, NULL},
                    case_stmts     : {cases, NULL},
                    default_stmt   : /statement/ >

goto_stmt     :   < goto_label     : ident >

label_stmt    :   < label          : ident,
                    stmt           : /statement/ >

dim_i         :   < array_dim      : {/expr/, NULL} >

init_val_i    :   < init_value     : /expr/,
                    level          : constant >

type_name     :   < typename       : type_spec,
                    abstract_var   : {abst_var, NULL},
                    struct_tag     : {st_tag, NULL} >

cases         :   < [case_stmt     : case_i] >

case_i        :   < case_expr      : /expr/,
                    stmt           : /statement/ >

abst_var      :   < is_pointer     : {true, NULL},
                    indirections   : {indir_level, NULL},
                    is_function    : {true, NULL},
                    isptr_to_fn    : {true, NULL},
                    isfn_ret_ptr   : {true, NULL},
                    is_array       : {true, NULL},
                    isary_of_ptrs  : {true, NULL},
                    dimensions     : {dims, NULL},
                    dimension_num  : {dim_num, NULL} >
```

# APPENDIX D

## SOURCE CODE FILES OF FRONT END

This appendix gives brief notes on the source code files of the front end developed in this thesis. This is given to aid those who carry further work.

| | |
|---|---|
| scanner.l | This file contains the lex specification which generated the lexer of the front end. |
| parser.y | This file contains the yacc specification which generated the parser of the front end. |
| tree | This file contains the treegen specification, which generated the tree builder, which provided the routines used in building the IR. |
| typecheck.c | This file has the C code which does the static type checking of the IR. |
| sym_tab.c | This file contains the C code for implementing the symbol table. |
| main.c | This file contains the main() function. |

other_fns.c This file has all other C code, which is part of this front end.

The header files which contain the definitions of variables and macro definitions are ext_defs.h, global_defs.h, operators.h, sym_tab.h, and temp_defs.h.